Sgap: Towards Efficient Sparse Tensor Algebra Compilation for GPU

Genghan Zhang[1], Yuetong Zhao[1], Yanting Tao[1], Zhongming Yu[2], Guohao Dai[3*], Sitao Huang[4], Yuan Wen[5], Pavlos Petoumenos[6] and Yu Wang[1*]

[1]Department of Electronic Engineering, Tsinghua University, Rhom 4101, Beijing, 100084, China.
[2]Department of Computer Science and Enigeering, University of California San Diego, Gilman Drive, La Jolla, 92093, CA, USA.
[3]Qingyuan Research Institute, Shanghai Jiao Tong University, Room 318A, Building A No. 930 Jianchuan Road, Shanghai, 200240, China.
[4]Department of Electrical Engineering and Computer Science, University of California Irvine, 3215 Engineering Hall, Irvine, 92697, CA, USA.
[5]Department of Computer Science, University of Aberdeen King's College,Meston Building, Aberdeen,AB24 3UE, United Kingdom.
[6]Department of Computer Science, University of Manchester, Kilburn Building, Manchester, M13 9PL, United Kingdom.

*Corresponding author(s). E-mail(s): daiguohao@sjtu.edu.cn; yu-wang@tsinghua.edu.cn;

Abstract

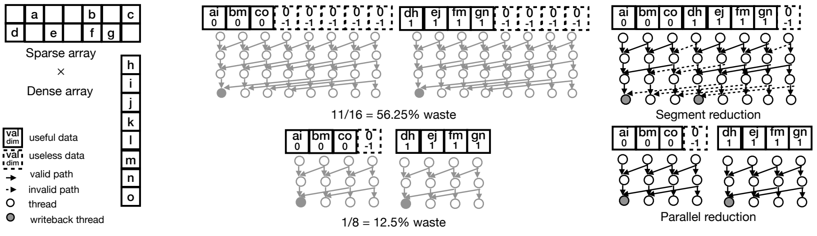
Sparse compiler is a promising solution for sparse tensor algebra optimization. In compiler implementation, **reduction** in sparse-dense hybrid algebra plays a key role in performance. Though GPU provides various reduction semantics that can better utilize the parallel computing and memory bandwidth capacity, the central question is: *how to elevate the **flexible reduction semantics** to sparse compilation theory that assumes serial execution*. Specifically, we have to tackle two main challenges: (1) there are wasted parallelism by adopting static synchronization granularity (2) static reduction strategy limits optimization space exploration. We propose Sgap: ***segment group*** and ***atomic parallelism*** to solve these problems. Atomic parallelism captures the flexible reduction semantics to systematically analyze the optimization space of sparse-dense hybrid algebra on GPU. It is a new optimization technique beyond current compiler-based and open-source runtime libraries. Segment group elevates the flexible reduction semantics to suitable levels of abstraction in the sparse compilation theory. It adopts changeable group size and user-defined reduction strategy to solve challenge (1) and (2), respectively. Finally, we use GPU sparse matrix-matrix multiplication (SpMM) on the TACO compiler as a use case to demonstrate the effectiveness of segment group in reduction semantics elevation. We achieve up to **1.2**× speedup over the original TACO's SpMM kernels. We also apply new optimization techniques found by atomic parallelism to an open-source state-of-the-art SpMM library dgSPARSE. We achieve **1.6**× ∼ **2.3**× speedup on the algorithm tuned with atomic parallelism.

Keywords: Sparse compiler, Sparse tensor algebra, SpMM, GPU

1 Introduction

Sparse tensor algebra has been widely used in many fields, including machine learning [1–3], data analysis [4], scientific computing [5, 6], graph processing [7]. However, it is challenging to optimize sparse tensor applications because of diversity in computation patterns and irregularity in memory access behavior. Sparse compilers have shown great potential to solve this problem. Sparse compilers can use **one** monolithic theory to express diverse data formats and operations, and provide flexible user interface, enabling users to explore the optimization space given data and hardware. Therefore, more and more researchers are turning to sparse compilers for general solutions [8–15].

However, it is challenging to design a sparse compiler that can both compile various algebras and generate highly optimized code. In particular, *sparse-dense hybrid algebra* on GPU brings unique challenges to sparse compilers. After analysing sparse-dense hybrid algebra's mathematical expression, we find out that **reduction** is its key operation [16–18]. There are several possible ways to do reduction on GPUs. Different reduction methods are preferred for different workloads. Choosing the correct reduction method can accelerate kernels [19, 20]. For example, controlled experiments in [19] show that parallel reduction can outperform conditional reduction and vice versa by 2× ∼ 4×. However, current sparse compilers lack the abstraction for such flexible reduction semantics. That is because they assume the code executes serially. GPU reduction is different from the serial reduction in that it changes the reduction code's structure (e.g., control-flow and loop basic block). Therefore, it cannot be naively generated by directly adding or replacing some instructions like the *unroll* in CPU. Solving this problem requires elevating reduction semantics to the sparse compilation theory in a systematic way.



(a) Example reduction **(b) Synchronization granularity** **(c) Reduction strategy**

Fig. 1 Sparse compilers suffer from static synchronization granularity and static reduction strategy. (a) Example reduction with legends in latter subfigures. (b) Parallelism waste caused by improper synchronization granularity. (c) One type of segment reduction and one type of parallel reduction. Segment reduction has two writeback threads and parallel reduction has one.

However, elevating the flexible reduction semantics to sparse compilation theory faces two main challenges: (1) **Static synchronization granularity wastes parallelism**: GPU synchronizes a group of threads whose group size is power of 2, which we term as synchronization granularity. Threads can pass local register values to another thread in the same group. However, static synchronization granularity may waste parallelism when inputs are dynamic. For example, if not all threads' register values are gathered, threads that do not influence the reduction result still have to wait to be synchronized. In other words, the synchronization granularity is too large for such input data, as is shown in Fig. 1 (b). However, current sparse compilers only assume synchronization granularity to be 32, which wastes the parallelism. This is the limitation of current sparse compilers. (2) **Static reduction strategy limits optimization space exploration**: GPU has provided very flexible methods to do reduction. Multiple threads in a thread group will write back to the final results. We name such thread *writeback thread*. There could be more than one writeback thread in a thread group. The thread indices of writeback threads can also be decided at runtime and are controlled by the reduction strategy. Different algorithms favor different reduction strategies. For example, as is shown in Fig. 1 (c), if we assign a given number of non-zeros to each thread group, it has to use segment reduction. That is because threads need to write back according to the coordinate and thus writeback thread is decided at runtime. However, in another algorithm where all threads in a group are guaranteed to write back to the same place, it can use parallel reduction [20]. However, current sparse compilers assume that only the first thread in a thread group is the writeback thread and use parallel reduction.

To tackle these challenges and build a more efficient sparse compiler, we propose ***atomic parallelism*** and ***segment group*** in this paper and implement our techniques in a real sparse compiler TACO [11, 21–23]. Atomic parallelism models the optimization space of sparse-dense hybrid algebra from the reduction view. It uses the minimal data and reduction parallelism to distinguish different algorithms of a given algebra. Minimal data are used to define reduction strategy and reduction parallelism for synchronization granularity.

We use this model to propose new optimization techniques. Segment group is a new abstraction for sparse compilation theory. It captures the dynamic synchronization granularity and dynamic reduction strategy. To be specific, we use flexible group size to solve challenge (1) and design full-stack support for user-defined reduction strategy, which solves challenge (2). As is shown in Fig. 2, segment group extends the expression ability of original sparse compilation theory. Finally, we use sparse matrix-matrix multiplication (SpMM) as an example to demonstrate atomic parallelism and segment group. SpMM is one of the most widely used sparse-dense hybrid algebra. It is the core operator of many emerging applications [24–27]. It is also the simplest form of sparse-dense hybrid algebra.

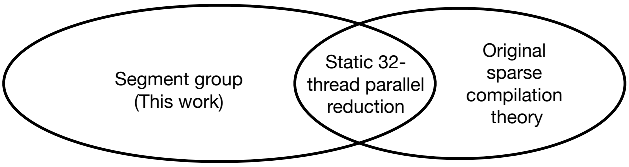


Fig. 2 Venn diagram for the relation between atomic parallelism and original sparse compilation theory. The element is the point in the algorithm design space of a sparse-dense hybrid algebra. Original sparse compilation theory can only express parallel reduction with group size 32. However, it can also express some optimization points, for example, loop reorder, beyond atomic parallelism. The union of segment group and original theory creates a new sparse compilation theory.

Therefore, this work manages to push the frontier a step forward on these two challenges by a combined method involving segment group and atomic parallelism which we called ***Sgap*** in this paper. Our contributions are as follows:

1. We propose a framework ***atomic parallelism*** to analyse sparse-dense hybrid algebra and propose new SpMM designs beyond previous works [17, 19, 28–30].
2. Based on the atomic parallelism, we point out that current sparse compilers miss important optimization opportunities. We propose a new abstraction ***segment group*** for sparse compilers. Segment group can reduce parallelism waste and improve workload balance.
3. We implement segment group in TACO and get up to 1.2× speedup on average over the original TACO's SpMM kernels. Next, we generalize our findings from TACO to dgSPARSE [19], an open-source state-of-the-art SpMM library. We achieve 1.6× $\sim$ 2.3× speedup over dgSPARSE on the algorithm we tune.

The rest of this paper is organized as follows. Background information is provided in Section 2. Section 3 introduces atomic parallelism and Section 4 is for segment group. Then the implementation of segment group in TACO is detailed in Section 5. After that, we illustrate the combination of atomic parallelism and segment group in TACO. Our evaluation of new SpMM algorithms in TACO and generalization to dgSPARSE is presented in Section 7. The paper is concluded in Section 8.

2 Background

2.1 Sparse-dense Hybrid Algebra

Sparse-dense hybrid algebra can be defined in two equivalent forms: the tensor formulation (TF) in Eq. 1 and the database formulation (DF) in Eq. 3. From TF sparse-dense hybrid algebra because the operands of it are sparse and dense, for example, MTTKRP (Matricized Tensor Times Khatri Rao Product) [16], SDDMM (Sampled Dense-Dense Matrix Multiplication) [31], SpMM (sparse Matrix-Matrix Multiplication) [17], TTM (Tensor Times Matrix Product) [18]. We use Einstein's summation to define sparse-dense hybrid algebra in AF as Eq. 1.

$$\mathbb{Y}_{y_1,y_2,\cdots,y_M} = \mathbb{A}_{a_1,a_2,\cdots,a_N} \prod_{i=1}^{D} \mathbb{X}^j_{x_1^j,x_2^j,\cdot,x^j_{M^j}} \tag{1}$$

$\mathbb{Y}$ is the output tensor, $\mathbb{X}^j$ are dense input tensors, and $\mathbb{A}$ is the sparse input tensor. At least one level a_N in $\mathbb{A}$ does not store in dense format. $y_1, y_2, \cdots, y_M$,$a_1, a_2, \cdots, a_N$,$x_1^j, x_2^j, \cdots, x^j_{M^j}$ are in the same index variable set. M is the mode of output tensor, and N is the mode of sparse input tensor. D is the number of dense input tensors, and M^j is the mode of dense input tensor $\mathbb{X}^j$. Specifically, MTTKRP, TTM, SDDMM, and SpMM are expressed as:

$$\mathbb{Y}_{i,j} = \mathbb{A}_{i,k,l}\mathbb{X}^1_{k,j}\mathbb{X}^2_{l,j} \tag{2a}$$

$$\mathbb{Y}_{i,j,l} = \mathbb{A}_{i,j,k}\mathbb{X}^1_{k,l} \tag{2b}$$

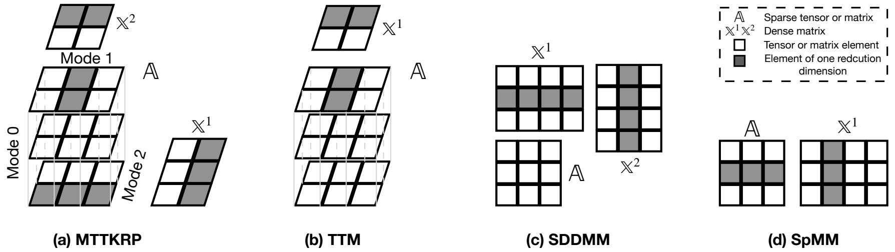
$$\mathbb{Y}_{i,k} = \mathbb{A}_{i,k}\mathbb{X}^1_{i,j}\mathbb{X}^2_{j,k} \tag{2c}$$

$$\mathbb{Y}_{i,k} = \mathbb{A}_{i,j}\mathbb{X}^1_{j,k} \tag{2d}$$

We use message-passing to define sparse-dense hybrid algebra in DF as Eq. 3.

$$Q(dst) = \oplus_{src \in Q_0(dst)}\{src, \otimes(Q_1(src, dst), Q_2(dst))\} \tag{3}$$

Q, Q_0, Q_1, Q_2 are queries for the relevant database. We follow the idea of logical-physical storage seperation [32]. The value of $Q(k)$ is defined as $Q(dst) = D(f(dst))$. D is the relevant database of Q, storing $(id, value)$ in ascending order of id, where $id \in \mathbb{Z}$ and $value \in \mathbb{R}^n$. dst is any hashable key and f is a function $K \to \mathbb{Z}$. $\oplus$ can be any commutative operation and $\otimes$ can be any function that takes two objects as input and output one object that can be operated by $\oplus$. The result of $\oplus$ is written to $f(dst)$ in Q. Sparse-dense hybrid algebra is sparse because $Q_0(dst)$ for all dst are diverse. In other words,



(a) MTTKRP (b) TTM (c) SDDMM (d) SpMM

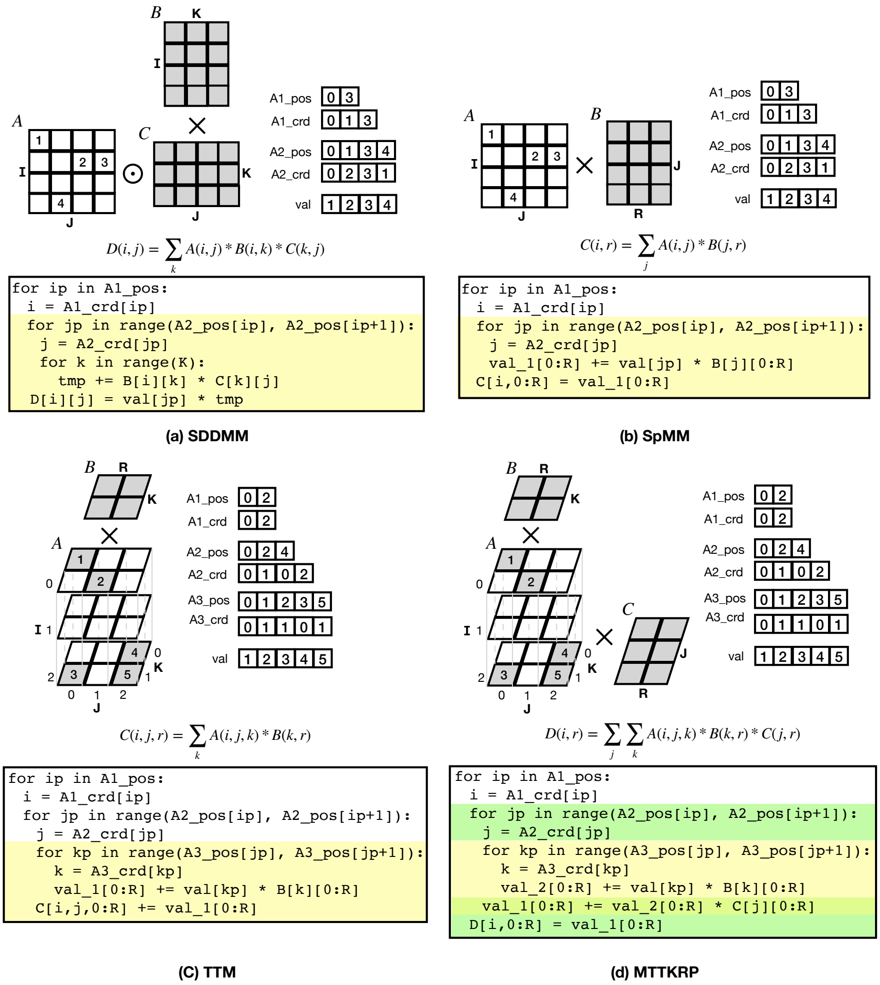
Fig. 3 Examples of sparse-dense hybrid algebra. The consecutive grey parallelograms or squares represent the reduction modes.

$Q_0(i) \bigcap Q_0(i+1) \sim \emptyset$. Such algebra is dense because values in D, D_1, D_2 are scalar, dense vectors, or dense matrices.

The core operation of sparse-dense hybrid algebra is *reduction* and reduction in different kernels behaves similarly. This key observation motivates atomic parallelism because we only need to optimize the common reduction operations and use the compiler to optimize different sparse-dense hybrid algebra kernels automatically. For example, in TF kernels do reduction on l, k dimensions in MTTKRP, k in TTM, j in SDDMM and SpMM. The reduction can be along one sparse and one dense dimension, as in MTTKRP, TTM, and SpMM. It can also be along two dense dimensions, as in SDDMM. Fig. 3 illustrates these examples and highlights the reduction dimensions. We also give concrete code examples in Fig. 4. It shows that some of these kernels share common reduction codes. For example, MTTKRP contains two reductions, each behaving the same as the reduction in SpMM. Such property can also be illustrated in DF. As shown in Fig. 5, for the first reduction, the value of D_1 both are scalar; the value of D_2 both are vectors. For the second reduction of MTTKRP, though the value of D_1 is a vector, which is different from SpMM's first reduction, $\oplus$ behaves the same because $\otimes$ here is element-wise vector product.

2.2 SpMM Optimization

As explained above, the reduction is the core operation of sparse-dense tensor algebra and some kernels share the same type of reduction. Without loss of generality, we take SpMM as an example to optimize the reduction in this paper. The optimization techniques can be easily generalized to expedite other sparse-dense hybrid algebra kernels. Yang et al. [28] selects between two algorithms to achieve respectively even distribution of nnz among parallel processors and row-splitting among threads. Adaptive Sparse Tiling (ASpT) [29] aims at improving data locality and thus reduces the total number of accesses to global memory. Ge-SpMM [17] proposes Coalesced Row Caching (CRC) method to enable coalesced memory access to both sparse and dense matrices and Coarse-grained Warp Merging (CWM) method for SpMM merging workloads from different warps to reuses loaded sparse matrix. Mehrabi et al. [30] proposes several row permutation strategies for CSR format to enhance load



$$D(i,j) = \sum_k A(i,j) * B(i,k) * C(k,j)$$

```
for ip in A1_pos:
  i = A1_crd[ip]
  for jp in range(A2_pos[ip], A2_pos[ip+1]):
   j = A2_crd[jp]
   for k in range(K):
     tmp += B[i][k] * C[k][j]
  D[i][j] = val[jp] * tmp
```

(a) SDDMM

$$C(i,r) = \sum_j A(i,j) * B(j,r)$$

```
for ip in A1_pos:
  i = A1_crd[ip]
  for jp in range(A2_pos[ip], A2_pos[ip+1]):
   j = A2_crd[jp]
   val_1[0:R] += val[jp] * B[j][0:R]
  C[i,0:R] = val_1[0:R]
```

(b) SpMM

$$C(i,j,r) = \sum_k A(i,j,k) * B(k,r)$$

```
for ip in A1_pos:
  i = A1_crd[ip]
  for jp in range(A2_pos[ip], A2_pos[ip+1]):
   j = A2_crd[jp]
   for kp in range(A3_pos[jp], A3_pos[jp+1]):
     k = A3_crd[kp]
     val_1[0:R] += val[kp] * B[k][0:R]
   C[i,j,0:R] += val_1[0:R]
```

(C) TTM

$$D(i,r) = \sum_j \sum_k A(i,j,k) * B(k,r) * C(j,r)$$

```
for ip in A1_pos:
  i = A1_crd[ip]
  for jp in range(A2_pos[ip], A2_pos[ip+1]):
   j = A2_crd[jp]
   for kp in range(A3_pos[jp], A3_pos[jp+1]):
     k = A3_crd[kp]
     val_2[0:R] += val[kp] * B[k][0:R]
   val_1[0:R] += val_2[0:R] * C[j][0:R]
  D[i,0:R] = val_1[0:R]
```

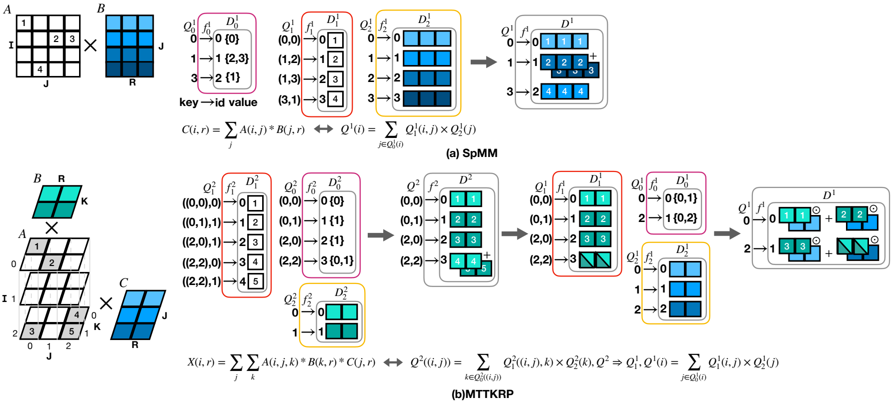
(d) MTTKRP

Fig. 4 Code examples of reduction in sparse-dense hybrid algebra in TF. The colored lines are reduction codes. MTTKRP has two levels of reduction, colored green and yellow, respectively. The overlapped region means that the first-level reduction's output serves as the second-level reduction's input. We follow the naming rules in [12] for the storage of A.

balance and data locality. DA-SpMM [19] is a data-aware kernel selector among 8 algorithms according to 3 dimensions in the space dealing with dynamic input data.

2.3 Sparse Compilers

The complexity of optimizing sparse tensor algebra comes from four directions: data, data format, algebra, and hardware. Researchers often develop a technique for one data format, one algebra, and one hardware. Such a library method heavily relies on experts and engineering work [33–35]. However, sparse



$$C(i,r) = \sum_j A(i,j) * B(j,r) \leftrightarrow Q^1(i) = \sum_{j \in Q_0^1(i)} Q_1^1(i,j) \times Q_2^1(j)$$

(a) SpMM

$$X(i,r) = \sum_j \sum_k A(i,j,k) * B(k,r) * C(j,r) \leftrightarrow Q^2((i,j)) = \sum_{k \in Q_0^2((i,j))} Q_1^2((i,j),k) \times Q_2^2(k), Q^2 \Rightarrow Q_1^1, Q^1(i) = \sum_{j \in Q_0^1(i)} Q_1^1(i,j) \times Q_2^1(j)$$

(b)MTTKRP

Fig. 5 Illustration of common reduction in MTTKRP and SpMM. The equivalent expressions of the same kernel in TF and DF are below each sub-figure.

compilers can extremely reduce such engineering burden and boost innovation in this area. Unlike the library method, sparse compilers aim to use **one** monolithic theory to express all data formats, all algebras, and provide flexible user interface, which enables users to explore the optimization space given data and hardware. Research on sparse compilers can be divided into two categories: (1) *Pass-oriented.* Given the imperative code, design compilation passes to optimize the code [8–10]. (2) *Language-oriented.* View sparse compiler as a programming language and design lowering and scheduling process [12, 14, 15]. Especially, TACO is a fundamental breakthrough on this problem. To the best of our knowledge, it is the first to propose a practical sparse compilation theory. MLIR sparse dialect [14] implements TACO's sparse compilation theory as MLIR dialect. SparseTIR [15] follows the design philosophy of TensorIR [36], but it still uses some of the TACO's concepts such as position and coordinate space. TACO also motivates innovations on accelerators for sparse tensor algebra [37].

2.4 TACO

TACO (The Tensor Algebra Compiler) is a fast and versatile compiler-based library for sparse linear and tensor algebra [11, 12, 21, 23]. TACO has three types of inputs: a tensor algebra expression (in an Einstein summation notation or reduction notation); level formats of input and output tensor; schedule commands. We will introduce TACO in the front-end, middle-end, and back-end order. The workflow of TACO is illustrated in Fig. 6.

2.4.1 Front-end

At the front-end, the tensor algebra expression is concretized to concrete index notation [21]. The concrete index notation (CIN) is a language that describes the execution of a tensor algebra. Unlike bare tensor algebra expression, CIN describes the loop, index variables relations, workspace, hardware platform,

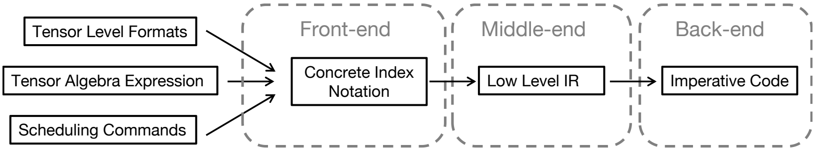


Fig. 6 Overview of the TACO workflow

etc. Schedule commands transform the CIN. For example, a *precompute* schedule will add a *where* statement to the CIN. Though TACO provides a clean and powerful scheduling API to transform CIN, the user can still change the CIN directly. TACO provides a match function that can take lambda expression as input. The function can modify CIN when it meets a specific type of CIN node or a pattern of CIN nodes. Moreover, users can define a child class of IndexNotationRewriter that can directly rewrite the CIN. Such technique is used to implement segment group.

2.4.2 Middle-end

At the middle-end, CIN will be transformed to imperative IR (or low level IR (LLIR)). LLIR describes the basic blocks, for example, for-loop, while-loop, and if-statement. LLIR is almost the executable code. The output of the middle-end is a chain of LLIR. The sparse iteration theory [12] guides the CIN to LLIR process. It ensures that different tensors only coiterate over elements that can generate non-zero output. Specifically, TACO designs lower functions for every statement of CIN and lattices in the sparse iteration space. However, current lower functions only assume serial reduction is done on the compressed level of sparse tensors. We will break the serial code assumption to implement segment group. Moreover, we suggest that more flexible or even user-defined lowerers should be designed in the future.

2.4.3 Back-end

At the back-end, LLIR will be transformed to code for different backends. In this paper, we target the CUDA code generation. TACO CUDA code generator has some assumptions that previous papers did not thoroughly explore. TACO deals with CUDA code generation in a nested loop favor [23]. Moreover, it only generates one dimension of block and thread. That is, it only has blockIdx.x and threadId.x. When the index variable of a for-loop LLIR is bound on the GPUBlock, it will use blockIdx.x to index this index variable. In the CPU case, it will emit a real for-loop. Such variable is assumed to increment by 1. Index variables bound on GPUWarp and GPUThread are assumed to be the outer and inner variables of threadIdx.x. The tile size depends on the index variable on GPUThread. The mixture of tiling and synchronization semantics of GPUWarp loses some optimization opportunities. We will discuss this later and improve it in our implementation.

3 Atomic parallelism

3.1 Computation unit model

We observe that the core operation of sparse-dense hybrid algebra is the reduction. Therefore, the core of our model is *how many data are reduced and are synchronized in which way.* We model the atomic computation unit as **thread**. A thread executes a serial program. All threads execute the same program independently with each own's input data and are distinguished by threadId. Threads can do synchronization in groups with *reduction parallelism* of 2,4,8,16, or 32. We model GPU computation as *unlimited* parallel threads and define the number of threads as *resource parallelism* that GPU can provide. We do not consider the shared memory, grid level, and the mapping of the thread block or the streaming processor. Instead, we view them as reasonable implementation details after the basic parallel pattern is decided. In other words, there can be many kinds of implementation for each algorithm in atomic parallelism. In this sense, atomic parallelism can encourage more GPU optimization innovation.

3.2 Overview of atomic parallelism

To define the parallel pattern concretely, we propose *atomic parallelism.* A program with *atomic parallelism* cannot be paralleled anymore. In other words, a thread at least executes the amount of data denoted by atomic parallelism. Formally, atomic parallelism is defined as the Cartesian product of *minimal data.* Minimal data is the minor data of one category a thread can execute. Atomic parallelism can be used to construct the optimization space of any sparse-dense hybrid algebra under the GPU model, but we focus on SpMM in this paper.

Indeed, tiling, manipulating shared memory, and thread mapping [17, 30, 38, 39] are also important for SpMM on GPU. They are crucial for SpMM, especially with many dense columns(usually more than 128 columns), because the computation will be more *workload* intensive and bounded by the memory access for dense columns. However, we focus on SpMM with fewer dense columns(usually less than 8 columns), which are more *balance* intensive and bounded by the maximum warp execution cycles.

SpMM has two orthogonal atomic parallelisms: minimal data can be (1) $\{\frac{1}{g}, 1, g\}$ non-zeros of the sparse matrix and $\{\frac{1}{c}, 1, c\}$ columns of the dense matrix; (2) $\{\frac{1}{g}, 1, g\}$ rows of the sparse matrix and $\{\frac{1}{c}, 1, c\}$ columns of the dense matrix. $c \in \mathbb{Z}^+$ and $g \in \mathbb{Z}^+$ are tunable parameters. Though they can be 1, they have different meanings from 1, because they are *tunable.* Therefore, the atomic parallelism space of SpMM is described in $< x\,nnz, y\,col >$ or $< x\,row, y\,col >$. Resource parallelism only multiplies one element of the atomic parallelism. For example, given resource parallelism r, the amount of executed data equals $< r \times x\,nnz, y\,col >$ or $< x\,nnz, r \times y\,col >$. Besides, a fractional amount of data means multiple threads may execute on the same datum.

For example, $< \frac{1}{g}\, row, 1\, col >$ means that g threads execute the same row collaboratively.

3.3 SpMM optimization space formalization

We use atomic parallelism and reduction parallelism $\{< ... >, r\}$ to define an SpMM kernel. $< ... > \in \{\frac{1}{g}, 1, g\}nnz \times \{\frac{1}{c}, 1, c\}col$ or $\{\frac{1}{g}, 1, g\}row \times \{\frac{1}{c}, 1, c\}col$. They describe the minimal data. And the *reduction parallelism* $r \in \{2, 4, 8, 16, 32\}$ assigns how many threads are synchronized each time. Fig. 7 illustrates the SpMM optimization space. However, not all points in the atomic parallelism space are legal in optimization space. Fig. 8 illustrates the details of space pruning. There are three rules for legal points:

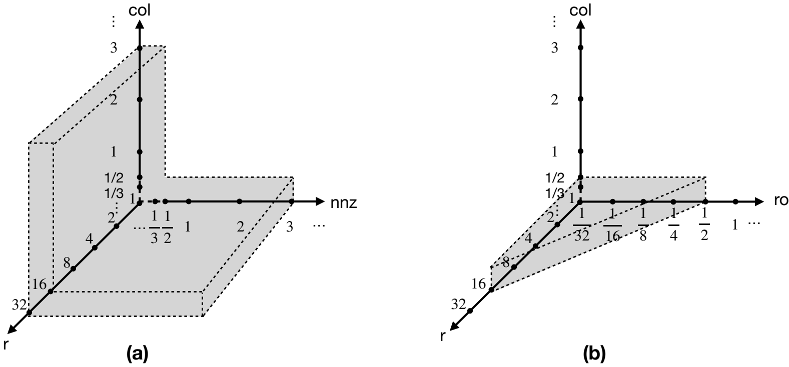


Fig. 7 SpMM optimization space. The grey area is illegal. The dashed line part of the axis represents hardware dependent end of the axis.

1. $\{< \frac{1}{g}\, nnz, x\, col >, r\}$, $\{< x\, nnz, \frac{1}{c}\, col >, r\}$ are illegal because one non-zero must by multiplied by at least one element in the dense matrix.
2. $\{< \frac{1}{g}\, row, x\, col >, r\}(\frac{r}{g} < 1)$ is illegal because parallel reduction only has one writeback thread.
3. $\{< \frac{1}{g}\, row, \frac{1}{c}\, col >, r\}$ is illegal because it conflicts with the rule that resource parallelism only multiplies one element of the atomic parallelism.

The state-of-the-art algorithm space, DA-SpMM [19] is in the atomic parallelism design space. It proposes a three-dimensional SpMM algorithm design space. We claim that the design space of DA-SpMM is included in the atomic parallelism space. To be specific, EB+PR is $\{< 1\, nnz, c\, col >, 32\}$, RB+PR is $\{< \frac{1}{32}\, row, c\, col >, 32\}$, EB+SR is $\{< 32\, nnz, c\, col >, 1\}$, and RB+SR is $\{< 1\, row, c\, col >, 1\}$. c means coarsen factor, g means group size. Though real CUDA code with $1\, row$ or $1\, nnz$ may have minimal data greater than one because of limited resource parallelism, we still label the algorithm as $1\, row$ or $1\, nnz$. The RM/CM is the implementation detail and is included in atomic parallelism in theory.

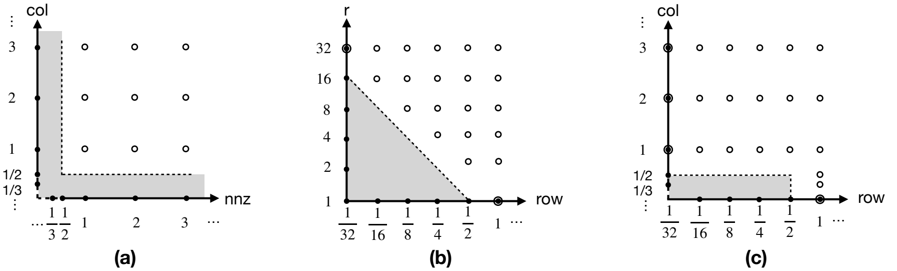
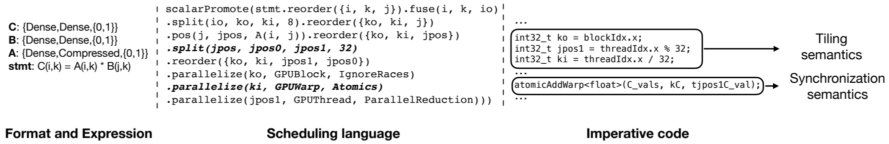


Fig. 8 Projections of SpMM optimization space. Grey areas are illegal and hollow circles are legal points. Sub-figures (a), (b), and (c) correspond to Rule 1, 2, and 3 respectively.

4 Segment group

4.1 Current warp-level abstraction

Current sparse tensor compilers with CUDA backend take warp as the rank of a thread (*tiling*), a particular parallel unit (*synchronization*) or just a hardware instruction. For example, TACO assumes warp and thread to be the outer and inner loop, and the *warpSize* depends on the split factor. It should be noted that no synchronization behavior is assumed in this case. TACO also takes the 32-thread warp reduction as atomic addition at the GPUWarp parallel unit and assumes users will split the last level loop with *warpSize* = 32. In this case, CUDA warp is taken as a for-loop with extent *warpSize* and incremental step 1. Then they will emit CUDA warp primitives such as *__shfl_down_sync* to do the reduction. Fig. 9 illustrates TACO's current GPU Warp semantics. On the contrary, TVM[40] only binds on thread and block level and does not assign any synchronization on the warp level. Instead, it takes 32 as a hardware feature and uses such intrinsic to fill in schedule parameters in auto-scheduler. Besides, it also uses warp as a memory load unit in TIR[40].


C: {Dense,Dense,{0,1}}
B: {Dense,Dense,{0,1}}
A: {Dense,Compressed,{0,1}}
stmt: C(i,k) = A(i,k) * B(j,k)

```
scalarPromote(stmt.reorder({i, k, j}).fuse(i, k, io)
.split(io, ko, ki, 8).reorder({ko, ki, j})
.pos(j, jpos, A(i, j)).reorder({ko, ki, jpos})
.split(jpos, jpos0, jpos1, 32)
.reorder({ko, ki, jpos1, jpos0})
.parallelize(ko, GPUBlock, IgnoreRaces)
.parallelize(ki, GPUWarp, Atomics)
.parallelize(jpos1, GPUThread, ParallelReduction)))
```

```
...
int32_t ko = blockIdx.x;
int32_t jpos1 = threadIdx.x % 32;
int32_t ki = threadIdx.x / 32;
...
atomicAddWarp<float>(C_vals, kC, tjpos1C_val);
...
```

Fig. 9 Tiling and synchronization semantics of GPU Warp in TACO

4.2 Overview of segment group

However, at least two existing assumptions should be improved for sparse compilers. First, the *tiling* and *synchronization* semantics of warp should be explicitly separated. As shown in atomic parallelism, the atomic and reduction parallelism can be different, and reduction parallelism is not necessarily 32. Second, synchronization semantics should be able to express various reduction strategies and flexible reduction granularity, instead of just parallel reduction for 32 threads. As shown in atomic parallelism, $\{< 1\,nnz, c\,col >, n\}$ requires synchronization of n threads with row number of their own. Therefore, the warp reduction should be able to reduce to different outputs instead of only one. Such change not only calls for changing the hand-coded warp level reduction functions but also for elevating the reduction pattern to higher-level compiler passes. Such semantics lifting calls for a new organization of basic blocks, new control flow, and new user-level APIs.

4.3 Relationship between segment group and atomic parallelism

Atomic parallelism models the optimization space of sparse-dense hybrid algebra from the reduction view. We use this model to propose new optimization techniques. As shown in Section 2, reduction is the key operation of sparse-dense hybrid algebra, which contains many different tensor algebras such as SpMM, SDDMM, MTTKRP, and TTM. Based on this observation, we define and explain segment group in Section 3, using SpMM as an example. We show that 3 opens new optimization space for SpMM. Such benefit can be generalized to other sparse-dense hybrid algebra. However, it requires repetitive engineering efforts to optimize case by case. In response to this issue, we propose segment group, a new abstraction for sparse compilers to ship performance benefits brought by atomic parallelism to users with only several lines of code changed on the user side.

In summary, we propose that sparse compilers for GPU should have abstraction segment group, that is, a *warp* that takes the *tiling* semantics, and a *group* that does different types of reduction *synchronization*. We will use TACO[1] to illustrate how to implement segment group, but other sparse compilers can also integrate segment group. Fig. 10 illustrates the workflow.

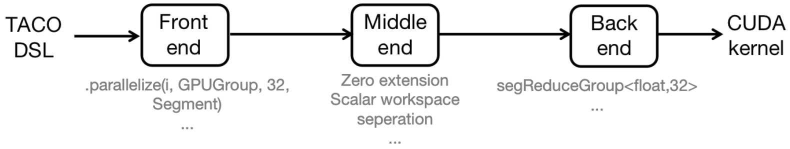


Fig. 10 Overview of segment group in the TACO workflow

5 Segment group for TACO

The original *parallelize* transformation is defined as *parallelize(IndexVar i, ParallelUnit pu, OutputRaceStrategy rs)* [23]. The transformation does parallel

[1] We build on commit d0654a8 https://github.com/zhang677/taco/tree/d0654a84137169883973c40a951dfdb89883fd9c

execution on IndexVar i, using ParallelUnit pu. And OutputRaceStrategy rs describes the data races during reductions. For GPU, pu can be GPUThread, GPUWarp, and GPUBlock. rs can be NoRaces, IgnoreRaces, and Atomics. We propose two new designs to TACO:

1. We add a new PrallelUnit, *GPUGroup*, to the *parallelize* transformation, and change the semantics of ParallelUnit *GPUWarp*.
2. We break the assumption that other transformations other than parallelize assumes serial code and design a new lower process to enable segment reduction.

5.1 New parallelize transformation

We assign the *tiling* semantics to GPUWarp and its *Atomic OutputRaceStrategy* will only serve to direct the lowering function instead of synchronization semantics. Because GPUWarp now only serves as the outer loop of tiling on threadIdx, it does not have *Atomic* semantics. Meanwhile, we add *GPUGroup* which has *ReductionStrategy* and *GroupSize* attributes instead of *OutputRaceStrategy. ReductionStrategy* describes the group's reduction type, and *GroupSize* assigns the reduction parallelism.

5.2 Reduction semantics elevation

TACO assumes that a sparse algebra compiler should do it best to ensure that only elements that can generate non-zero output will be calculated [12]. However, we point out that this assumption is not necessarily valid. The previous assumption is the best option for performance because the sparse iteration space theory is built on the assumption that the code runs serially. For CUDA code, however, such assumption is broken, which we term as *zero extension.* Zero extension means that some "out-of-bound" reduction can be allowed in the sparse iteration theory because it can later be executed by some warp primitives faster than for-loop.

5.3 Segment reduction lowering

```
//Original CUDA code
for(k=0;k<B2_dimension;k++){
  pA2_begin=i_blockStarts[block];
  pA2_end=i_blockStarts[block+1];
  fposA=block*256;
  i_pos=taco_binarySearchBefore(
  A2_pos,pA2_begin,pA2_end,fposA);
  i=i_pos;
  fposA=block*256+fpos1;
  if(fposA>=A2_pos[A1_dimension])
    break;
  f=A2_crd[fposA];
  kB=f*B2_dimension+k;
  while(fposA==A2_pos[i_pos+1]){
    i_pos=i_pos+1;
    i=i_pos;
  }
  kC=i*C2_dimension+k;
  float val=0.0;
  val=A_vals[fposA]*B_vals[kB];
  atomicAdd(&C_vals[kC],val);
}
```

Listing 1 Original CUDA code

```
//Modified CUDA code
for(k=0;k<B2_dimension;k++){
  pA2_begin=i_blockStarts[block];
  pA2_end=i_blockStarts[block+1];
  fposA=block*256+fpos1;
  i_pos=taco_binarySearchBefore(
  A2_pos,pA2_begin,pA2_end,fposA);
  i=i_pos;
  float val=0.0;
  if(fposA>=A2_pos[A1_dimension])
    val=0;
  else{
    f=A2_crd[fposA];
    kB=f*B2_dimension+k;
    while(fposA==A2_pos[i_pos+1]){
      i_pos=i_pos+1;
      i=i_pos;
    }
    val=A_vals[fposA]*B_vals[kB];
  }
  kC=i*C2_dimension+k;
  segReduceWarp<float,32>(C_vals,
  kC,val);
}
```

Listing 2 Modified CUDA code

Listing 1 and Listing 2 show the difference between codes generated by the original TACO and the modified TACO. They use the same schedule, except that code on the right uses segment reduction of GPUGroup with size 32.

scalar workspace. TACO assumes that the statement and the assignment of *scalar workspace* [21] are in the same basic block. However, this assumption is so strong that it restricts the expressive power of TACO. For example, in $\{< 1\,nnz, c\,col >, 32\}$ the scalar workspace should be assigned in a basic block belonging to an *else* but stated in the same context with reduction of scalar workspace, outside the assignment basic block.

Macro instruction. It is important to emit code in a modular way. Therefore, we design two new *macro instructions atomicAddGroup<T,G>(T* array, int idx, T value)* and *segReducWarp<T,G>(T* array, int idx, T value)*. They are template device functions that takes in the output array, the index of the output and the value reduced to the output[2]. They will do some kind of reduction on G threads, and G equals *GroupSize*. They will be stated in the header file and used as macro instructions in the final CUDA code. In fact, we borrow the *group* concept from the *cooperative group* in CUDA. Since CUDA 11.0, it has supported an easy-to-use API called cooperative group [3] that makes it only one-line-code effort to change reduction granularity to less than 32 threads.

[2] We do not actually integrate these macro instructions into TACO, because it is fairly straightforward and purely engineering. When testing the kernels, we just replace the atomicAdd with the new macro instructions. We open-source the modified TACO https://github.com/zhang677/taco/tree/parallelreduction.

[3] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups

6 TACO's support for four SpMM algorithms

This section will illustrate the atomic parallelism design space and our implementation of segment group. We first reexamine two SpMM algorithms proposed by TACO [23]. They use TACO to generate $\{< g\,nnz, c\,col >, 1\}$ and $\{< x\,row, c\,col >, 1\}$. We then use another two examples, $\{< \frac{1}{g}\,row, c\,col >, r\}$ and $\{< 1\,nnz, c\,col >, r\}$ to illustrate how the CIN is changed. The tensor algebra expression is $C(i,k) = A(i,j) * B(j,k)$. A's first level is dense and the second level is compressed. B and C are both dense matrices. A, B, and C all are row-major. We assume $N = 4$ and that thread per block (resource parallelism p) equals 256. We explicitly fill p, g, N, c into the CIN to show their arithmetic relations with CIN parameters. The actual CIN will not have undetermined variables.

6.1 TACO SpMM reexamination

Currently, TACO supports two algorithms in atomic parallelism. They don't need synchronization semantics and only tune on the tiling semantics. The implementation by TACO is shown in Listing 3 and 4. They force the synchronization granularity to be 1 which presents limited capability in reduction.

Concrete Index Notation for $\{< g\,nnz, c\,col >, 1\}$ is :

```
suchthat(forall(block,forall(warp,forall(thread,
forall(dense_val,where(C(i,k)+=tnnzC,forall(nnz,
tnnzC+=A(i,j)*B(j,k)))),GPUThread,Atomics),
GPUWarp,NoRaces),GPUBlock,NoRaces),
fuse(i,j,f) and pos(f,fpos,A(i,j)) and
split(fpos,block,fpos1,(p*g/(N/c))) and
split(fpos1,warp,nnz,g) and split(k,ko,thread,c)
and bound(ko,dense_val,N/c,MaxExact))
```

Listing 3 CIN for $\{< g\,nnz, c\,col >, 1\}$

Actually, TACO's *precompute* schedule fails to generate this CIN, so we use the IndexNotationRewriter technique mentioned in section 2.4.1 to get the CIN above. In the evaluation section of [23] it assumes $N = 128, g = 16, c = 4, p = 512$, which is a point in the $\{< g\,nnz, c\,col >, 1\}$.

Concrete Index Notation for $\{< g\,row, c\,col >, 1\}$ is :

```
suchthat(forall(block,forall(warp,forall(row,
forall(thread,forall(col,where(C(i,k)+=tjC,
forall(j,tjC+=A(i,j)*B(j,k)))),GPUThread,NoRaces)),
GPUWarp,NoRaces),GPUBlock,NoRaces),split(i,block,io,
p*g/(N/c))and split(io,warp,row,g) and split(k,ko,col,c)
and bound(ko,thread,N/c,MaxExact))
```

Listing 4 CIN for $\{< g\,row, c\,col >, 1\}$

The generated code can be directly executed. In the evaluation section of [23] it assumes $N = 128, g = 1, c = 4, p = 512$, which is also a point in the

$\{< g\,nnz, c\,col >, 1\}$. These two algorithms only use the *tiling* semantics of GPUWarp.

6.2 Two new algorithms

We introduce two algorithms to overcome the restricted scheme forced by TACO to improve workload balance. The algorithms provide functionality to change group size and reduction strategy through tuning nnz and rows. Listing 5 and 6 show the implementation.

Concrete Index Notation for $\{< \frac{1}{g}\,row, c\,col >, r\}$ is :

```
suchthat(forall(ko,forall(warp,forall(kii,where(C(i,k)+=tjpos1C,
forall(jpos1, forall(jpos0,tjpos1C+=A(i,j)*B(j,k)),GPUThread,
ParallelReduction))),GPUWarp, Atomics),GPUBlock,NoRaces),
fuse(i,k,io) and split(io,ko,ki,c*p/g) and split(ki, warp,kii,c)
and pos(j,jpos,A(i,j)) and split(jpos,jpos0,jpos1,g) and
parallelize(jpos1,GPUGroup,r,Atomics))
```

Listing 5 CIN for $\{< \frac{1}{g}\,row, c\,col >, r\}$

We find that TACO can support $g = 32, r = 32$, but it is not explored in the autoscheduling paper[4]. GPUGroup is bound on the indexVar that does the reduction. Generated macro-instruction, *atomicAddWarp<Type>*, is changed to *atomicAddGroup<Type, G>* to enable more fine-grained thread synchronization.

Concrete Index Notation for $\{< 1\,nnz, c\,col >, r\}$ is :

```
suchthat(forall(block,forall(warp,forall(ki,forall(fpos1,where(
C(i,k)+=tmp,tmp=A(i,j)*B(j,k)),GPUThread,Atomics)),GPUWarp,NoRaces),
GPUBlock,IgnoreRaces),fuse(i,j,f) and pos(f,fpos,A(i,j)) and
split(fpos,block,fpos1,p/(N/c)) and split (k,ko,ki,c) and bound(ko,
warp,N/c,MaxExact) and parallelize(jpos1,GPUGroup,r,Segment))
```

Listing 6 CIN for $\{< 1\,nnz, c\,col >, r\}$

This algorithm has no counterpart in the original TACO. We change the originally emitted *atomicAdd* to *segReduceGroup<Type,G>*, and the grouped segment reduction is done in the macro instruction. The lowerer of scalar workspace is changed to emit the code ready for segmented reduction.

7 Evaluation

Experiment settings. We evaluate the implementation and the generalization on three architectures:

- NVIDIA RTX 3090. Compute Capability 8.6 (68 Ampere SMs at 1.395 GHz, 24 GB GDDR6x, 936 GB/s bandwidth).

[4] [23]'s authors shared their code with us. We also use a similar code base to test our kernels in Section 7

- NVIDIA RTX 2080. Compute Capability 7.5 (46 Turing SMs at 1.515 GHz, 8 GB GDDR6, 448 GB/s bandwidth).
- NVIDIA Tesla V100. Compute Capability 7.0 (80 Volta SMs at 1.370 GHz, 16 GB HBM2, 900 GB/s bandwidth).

We use NVCC 11.6 and CUDA 11.6 with the same compilation flags as [23] when testing TACO and the same compilation flag as [19] when testing the generalized tuning. We carry 25 tests for each kernel to get the average execution time when evaluating TACO's generated CUDA kernels. We use nsight-compute[5] to get the execution time of tuned dgSPARSE kernels. We use the same sparse matrices as [19]. We evaluate on three different architectures to show that our techniques are not limited to specific traits on certain generations of GPU, but are valid on common SIMT architectures.

7.1 Performance of two new algorithms for TACO

This experiment aims to prove that segment group can improve the sparse compiler's expression ability and boost the performance of SpMM kernels generated by TACO. The dense input matrices have $N = 4$[6].

Against the static group size 32. We use $\{< \frac{1}{g}\,row, c\,col >, r\}$ to show the improvement brought by flexible group size r. Current TACO only supports $g = 32, r = 32$, so we keep the same g with TACO and change r. In Table 1 we show that $r = 8$ and $r = 4$ can bring over 2.0x speedup on average. We also measure the *normalized speedup*. Normalized speedup of A over B means that if A performs better than B, we count the speedup; otherwise, we assume the user can choose the better algorithm, and the speedup is counted as 1.

Table 1 Flexible group size speedup

Hardware	$r = 8$	$r = 8$ norm	$r = 4$	$r = 4$ norm
RTX 2080	2.451	2.478	2.456	2.483
RTX 3090	2.236	2.284	2.259	2.307
Tesla V100	2.086	2.143	2.094	2.150

Against the original reduction. We use $\{< 1\,nnz, c\,col >, r\}$ to illustrate the speedup brought by flexible reduction. Because they have different data types (nnz vs. row), we control c and r, and compare the execution of $\{< 1\,nnz, c\,col >, r\}$ with the best g configuration of $\{< \frac{1}{g}\,row, c\,col >, r\}$ each dataset. We only do this experiment on RTX 3090 and record the normalized speedup here. In Table 2 we show that segment reduction can bring up to 1.3x speedup over atomicWarp reduction. Limited by the number of threads per warp in GPU, r can only be $1, 2, 4, 8, 16, 32$. Therefore, users can try these values to tune r in practice.

[5] https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[6] We open source the testing code at https://github.com/zhang677/segTACO.

Table 2 Segment reduction normalized speedup

c	r=4	r=8	r=16	r=32
1	1.008	1.025	1.085	1.272
2	1.019	1.045	1.102	1.291
4	1.063	1.095	1.205	1.381

Against the original TACO SpMM algorithms. In this experiment, we compare the performance between TACO's original SpMM algorithms $\{< g\,nnz, c\,col >, 1\}$ and $\{< x\,row, c\,col >, 1\}$ [23] and two algorithms proposed by us, $\{< \frac{1}{g}\,row, c\,col >, r\}$ and $\{< 1\,nnz, c\,col >, r\}$. We assign reasonable values to g, c, x,and r, and tune these parameters. We record the best performance of each algorithm on each dataset. From Table 3 we conclude that segment group brings 1.1x∼1.2x normalized speedup. Fig. 11 shows the detailed data.

Table 3 Normalized performance of new algorithms

	RTX 3090	RTX 2080	Tesla V100
Speedup	1.191	1.098	1.223

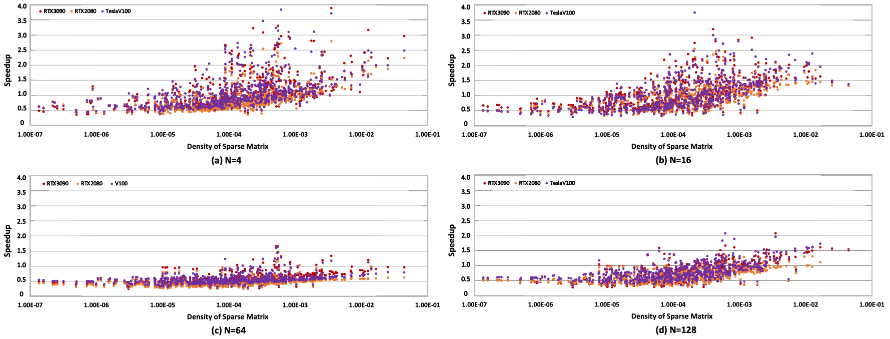


Fig. 11 Newly generated SpMM kernels performance compared with original TACO's best SpMM kernel for different number of dense matrix columns N. Density is defined as the number of non-zeros divided by the multiplication of the number of rows and cols for sparse matrix.

7.2 Generalization of atomic parallelism

In this experiment, we implement our atomic parallelism to dgSPARSE library [7], an open-source state-of-the-art SpMM and SDDMM library. We

[7] https://github.com/dgSPARSE

achieve up to 2.7x speedup on a certain SpMM algorithm. We keep the same sparse input matrix format (CSR) with dgSPARSE. After profiling, we find that row-major algorithms consistently outperform the col-major algorithms. Therefore, we target row-major. We are left with 4 algorithms: EB+SR+RM, EB+PR+RM, RB+SR+RM, RB+PR+RM. We will introduce the details of tuning RB+PR+RM and show the speedup.

To tune an actual GPU SpMM kernel, we require more fine-grained parameters than those in atomic parallelism. Parallelism is now two-fold: block-level and thread-level, instead of homogeneous threads. Besides, the memory hierarchy, such as the shared memory should be considered. Moreover, parallelism is limited in the physical world. For example, the largest thread-level parallelism is 1024 because a block has at most 1024 threads. The largest block-level parallelism is also finite(less than $2^{32} - 1$). GridSize can be arbitrary because the extra blocks will be taken care of by GPU scheduler.

Tuning parameters for RB+PR+RM can be divided into two categories. The first is how many workers are assigned to process one chunk of data. The second is how many chunks of data are assigned to one worker. RB+PR+RM has 7 tunable parameters. A block process *tileSz* real columns. *workerSz* threads process one vectorized column and *threadRw* sparse rows. *groupSz* threads are synchronized. *blockSz* denotes the number of threads per threadblock. *workerDimR* denotes the block parallelism of sparse rows. A vectorized column has *coarsenSz* consecutive real columns. If the overall sparse row parallelism is less than the number of rows in the sparse matrix, one thread may process more than one row. The tiling is“Dense major”; dense columns are fully parallelized. Specifically, *blockDim.x = min(N, tileSz) / coarsenSz * workerSz*. Full source parallelism of one block is *max(blockSz, blockDim.x * 2)*. In the dgSPARSE implementation, $tileSz = workerSz = groupSz = 32$, workerDimR equals the number of rows of the sparse matrix , $threadRw = 1$, $blockSz = 256$, and *coarsenSz=(N%4==0)?4:(N%2==0)?2:1.*

Based on the insights of this paper, we should separate tiling and synchronization, add finer-grained parallelism, and more flexible workload of each thread. Therefore, we propose to tune four parameters: $< groupSz, blockSz, tileSz, workerDimR >$. Actually, workerDimR can be arbitrary. However, we set it to be power of 2 or reciprocal power of 2 times of the original value in order to explore the local area in the design space. As in atomic parallelism we set groupSz as 2,4,8,16, or 32. tileSz is power of 2 larger than groupSz, and depends on N. blockSz is set 128,256, or 512 which are common values for the number of threads per threadblock. We tune the RB+PR+RM kernel for $N = 4, 16, 64, 128$. From Table 4 we conclude that tuning can bring 1.6x∼2.3x speedup over the original implementation[8].

Because DA-SpMM introduces a decision tree model to choose the best configuration for a given sparse matrix, we further explore the maximum

[8] We open source our implementation at https://github.com/dgSPARSE/dgSPARSE-Library/commit/9e3e4c18f40e76b97a805b8a9733258f7e9edeb6.

Table 4 Speedup over original implementation

Hardware	geomean[1]	max	N
RTX 3090	2.295	4.316	128
	2.181	4.432	64
	1.997	4.271	16
	2.046	7.819	4
RTX 2080	1.938	4.379	128
	1.927	4.430	64
	1.995	5.019	16
	2.307	8.582	4
Tesla V100	1.874	3.724	128
	1.824	3.846	64
	1.693	3.388	16
	1.852	6.114	4

[1] We use geometric mean to reduce outlier bias.

speedup that dynamic choices can bring. This experiment examines the necessity of designing a new model to choose the best parameters. From Table 5 we conclude that the most significant speedup of dynamic choices is 1.1x∼1.4x.

Table 5 Speedup over static implementation

Hardware	geomean	N	Best static
RTX 3090	1.124	128	$< 8, 256, 8, 1/2 >$
	1.114	64	$< 4, 256, 8, 1/2 >$
	1.310	16	$< 8, 256, 8, 1/2 >$
	1.406	4	$< 8, 256, 8, 1 >$
RTX 2080	1.095	128	$< 4, 256, 8, 1/2 >$
	1.114	64	$< 4, 256, 8, 1/2 >$
	1.276	16	$< 4, 256, 8, 1/2 >$
	1.310	4	$< 4, 256, 8, 1/2 >$
Tesla V100	1.137	128	$< 8, 256, 8, 1/2 >$
	1.177	64	$< 8, 256, 8, 1/2 >$
	1.367	16	$< 8, 256, 8, 1 >$
	1.326	4	$< 8, 256, 8, 1 >$

8 Conclusion

We propose atomic parallelism to analyze sparse-dense hybrid algebra and propose new SpMM designs. Based on atomic parallelism propose a new abstraction segment group to sparse compilers and remedy the missing optimization opportunities. First, we implement the new abstraction in TACO and achieve up to 1.2x speedup over TACO's original SpMM kernels. Then, we use atomic parallelism to tune an SpMM algorithm in dgSPARSE and get 1.6x∼2.3x speedup on the tuned algorithm. In the future, atomic parallelism can be exposed as an auto-tuning API for users to explore different synchronization granularity and reduction strategy for sparse-dense hybrid algebra.

References

[1] Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)

[2] Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)

[3] Liu, B., Wang, M., Foroosh, H., Tappen, M., Pensky, M.: Sparse convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 806–814 (2015)

[4] Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM review **51**(3), 455–500 (2009)

[5] Shantharam, M., Srinivasmurthy, S., Raghavan, P.: Characterizing the impact of soft errors on iterative methods in scientific computing. In: Proceedings of the International Conference on Supercomputing, pp. 152–161 (2011)

[6] Bell, N., Dalton, S., Olson, L.N.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing **34**(4), 123–152 (2012)

[7] Yuster, R., Zwick, U.: Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In: SODA, vol. 4, pp. 254–260 (2004). Citeseer

[8] Bik, A.J., Wijshoff, H.A.: Compilation techniques for sparse matrix computations. In: Proceedings of the 7th International Conference on Supercomputing, pp. 416–424 (1993)

[9] Venkat, A., Hall, M., Strout, M.: Loop and data transformations for sparse matrix code. ACM SIGPLAN Notices **50**(6), 521–532 (2015)

[10] Strout, M.M., Hall, M., Olschanowsky, C.: The sparse polyhedral framework: Composing compiler-generated inspector-executor code. Proceedings of the IEEE **106**(11), 1921–1934 (2018)

[11] Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. Proc. ACM Program. Lang. **1**(OOPSLA), 77–17729 (2017). https://doi.org/10.1145/3133901

[12] Kjolstad, F.: Sparse tensor algebra compilation. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA (Feb 2020). http://tensor-compiler.org/files/kjolstad-phd-thesis-taco-compiler.pdf

[13] Popoola, T., Shankar, R., Rift, A., Singh, S., Davis, E.C., Strout, M.M., Olschanowsky, C.: An object-oriented interface to the sparse polyhedral library. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), pp. 1825–1831 (2021). IEEE

[14] Bik, A.J., Koanantakool, P., Shpeisman, T., Vasilache, N., Zheng, B., Kjolstad, F.: Compiler support for sparse tensor computations in mlir. arXiv preprint arXiv:2202.04305 (2022)

[15] Ye, Z., Lai, R., Shao, J., Chen, T., Ceze, L.: Sparsetir: Composable abstractions for sparse compilation in deep learning

[16] Nisa, I., Li, J., Sukumaran-Rajam, A., Vuduc, R., Sadayappan, P.: Load-balanced sparse mttkrp on gpus. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 123–133 (2019). IEEE

[17] Huang, G., Dai, G., Wang, Y., Yang, H.: Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2020). IEEE

[18] Kurt, S.E., Raje, S., Sukumaran-Rajam, A., Sadayappan, P.: Sparsity-aware tensor decomposition. In: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 952–962 (2022). IEEE

[19] Dai, G., Huang, G., Yang, S., Yu, Z., Zhang, H., Ding, Y., Xie, Y., Yang, H., Wang, Y.: Heuristic adaptability to input dynamics for spmm on gpus. arXiv preprint arXiv:2202.08556 (2022)

[20] Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11 (2009)

[21] Kjolstad, F., Ahrens, P., Kamil, S., Amarasinghe, S.: Tensor algebra compilation with workspaces, 180–192 (2019)

[22] Chou, S., Kjolstad, F., Amarasinghe, S.: Format abstraction for sparse tensor algebra compilers. Proc. ACM Program. Lang. **2**(OOPSLA), 123–112330 (2018). https://doi.org/10.1145/3276493

[23] Senanayake, R., Hong, C., Wang, Z., Wilson, A., Chou, S., Kamil,

S., Amarasinghe, S., Kjolstad, F.: A sparse iteration space transformation framework for sparse tensor algebra. Proc. ACM Program. Lang. **4**(OOPSLA) (2020). https://doi.org/10.1145/3428226

[24] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J.: Eie: Efficient inference engine on compressed deep neural network. ACM SIGARCH Computer Architecture News **44**(3), 243–254 (2016)

[25] Wang, Z., Wohlwend, J., Lei, T.: Structured pruning of large language models. arXiv preprint arXiv:1910.04732 (2019)

[26] Lin, C.-Y., Luo, L., Ceze, L.: Accelerating spmm kernel with cache-first edge sampling for graph neural networks. arXiv preprint arXiv:2104.10716 (2021)

[27] Asgari, B., Hadidi, R., Cao, J., Lim, S.-K., Kim, H., *et al.*: Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In: 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), pp. 908–920 (2021). IEEE

[28] Yang, C., Buluç, A., Owens, J.D.: Design principles for sparse matrix multiplication on the gpu. In: European Conference on Parallel Processing, pp. 672–687 (2018). Springer

[29] Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., Sadayappan, P.: Adaptive sparse tiling for sparse matrix multiplication. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 300–314 (2019)

[30] Mehrabi, A., Lee, D., Chatterjee, N., Sorin, D.J., Lee, B.C., O'Connor, M.: Learning sparse matrix row permutations for efficient spmm on gpu architectures. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 48–58 (2021). IEEE

[31] Yu, Z., Dai, G., Huang, G., Wang, Y., Yang, H.: Exploiting online locality and reduction parallelism for sampled dense matrix multiplication on gpus. In: 2021 IEEE 39th International Conference on Computer Design (ICCD), pp. 567–574 (2021). IEEE

[32] Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM **13**(6), 377–387 (1970)

[33] Guennebaud, G., Jacob, B., et al.: Eigen. URl: http://eigen. tuxfamily. org **3** (2010)

[34] Naumov, M., Chien, L., Vandermersch, P., Kapasi, U.: Cusparse library. In: GPU Technology Conference (2010)

[35] Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel math kernel library. In: High-Performance Computing on the Intel® Xeon Phi™, pp. 167–188. Springer, ??? (2014)

[36] Feng, S., Hou, B., Jin, H., Lin, W., Shao, J., Lai, R., Ye, Z., Zheng, L., Yu, C.H., Yu, Y., et al.: Tensorir: An abstraction for automatic tensorized program optimization. arXiv preprint arXiv:2207.04296 (2022)

[37] Qin, E., Garg, R., Bambhaniya, A., Pellauer, M., Parashar, A., Rajamanickam, S., Hao, C., Krishna, T.: Enabling flexibility for sparse tensor acceleration via heterogeneity. arXiv preprint arXiv:2201.08916 (2022)

[38] Hidayetoğlu, M., Pearson, C., Mailthody, V.S., Ebrahimi, E., Xiong, J., Nagi, R., Hwu, W.-m.: At-scale sparse deep neural network inference with efficient gpu implementation. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2020). IEEE

[39] Xin, J., Ye, X., Zheng, L., Wang, Q., Huang, Y., Yao, P., Yu, L., Liao, X., Jin, H.: Fast sparse deep neural network inference with flexible spmm optimization space exploration. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2021). IEEE

[40] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., *et al.*: Tvm: An automated end-to-end optimizing compiler for deep learning. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pp. 578–594 (2018)